

# Matching Logic

## Grigore Rosu and Andrei Stefanescu (UIUC)

## Highlights

### Matching Logic = Operational Semantics + FOL

- A logic for reasoning about configurations
- Formulae
  - FOL over configurations, called **patterns**
  - Configurations are allowed to contain variables
- Models
  - Ground configurations
- Satisfaction
  - **Matching** for configurations, plus FOL for the rest

## Examples of Patterns

- $x$  points to sequence  $A$ , and the reversed sequence  $A$  has been output
  - $\exists a \exists p \exists \sigma \exists \omega \ (\langle x \mapsto a, p \rangle_{\text{env}} \langle \text{list}(a, A), \sigma \rangle_{\text{heap}} \langle \omega, \text{rev}(A) \rangle_{\text{out}})$
- **untrusted()** can only be called from **trusted()**
  - $\exists s_1 \exists s_2 \ (\langle \text{untrusted}() \rangle_k \langle s_1, \text{trusted}(), s_2 \rangle_{\text{fstack}})$
- Read/Write datarace (simplified)
  - $\exists X \exists a \ (\langle X \dots \rangle_k \langle X = a \dots \rangle_k)$

## Partial Correctness

- We have two rewrite relations on configurations
  - $\rightarrow$  given by the language operational semantics; **safe**
  - $\Rightarrow$  given by specifications; **unsafe**, has to be proved
- Idea (simplified for deterministic languages):
  - Pick **left**  $\rightarrow$  **right**. Show that always **left**  $\Rightarrow$  **( $\rightarrow \cup \Rightarrow$ )\* right** modulo matching logic reasoning (between rewrite steps)
- Theorem (soundness):
  - If **left**  $\rightarrow$  **right** and “**config matches left**” such that **config** has a normal form for  $\rightarrow$ , then “**nf(config) matches right**”

## Program Verification with MatchC

### Using standard I/O

```
void readWriteBuffer(int n)
/*@ rule <<> $ => return; </>
  <in> A => epsilon </in>
  <out>_epsilon => rev(A) </out>
  if n = len(A) w/
{
  int i;
  struct ListNode *x;
  i = 0;
  x = 0;
  /*@ inv <in> 7B </in> <heap> list(x)(7A) </heap>
  /*@ inv /\ i <= n /\ len(7B) = n - i /\ A = rev(7A) @ 7B w/
  while (i < n) {
    struct ListNode *y;
    y = x;
    x = (struct ListNode*) malloc(sizeof(struct ListNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1;
  }
  /*@ inv <out>_7A </out> <heap> list(x)(7B) </heap> /\ A = rev(7A @ 7B)
  while (x) {
    struct ListNode *y;
    y = x->next;
    printf("%d ", x->val);
    free(x);
    x = y;
  }
}
```

```
Terminal — bash — 76x14
mobile-255-135:matchC andrei$ time (gcc demo/2-io/io3.c ; echo '5 1 2 3 4 5
5 6 7 8 9 10' | ./a.out ; rm a.out )
1 2 3 4 5 10 9 8 7 6
real    0m0.035s
user    0m0.018s
sys     0m0.012s
mobile-255-135:matchC andrei$ ./matchC demo/2-io/io3.c
Loading Maude ..... DONE! [0.281s]
Verifying program ... DONE! [0.183s]
Verification succeeded! [123418 rewrites, 6 feasible and 4 infeasible paths]
Output: 1 2 3 4 5 10 9 8 7 6
mobile-255-135:matchC andrei$
```

### List manipulating functions

```
struct ListNode {
  int val;
  struct ListNode *next;
};

struct ListNode* reverse(struct ListNode *x)
/*@ rule <<> $ => return p1; </>
  <heap> list(x)(A) => list(p1)(rev(A)) </heap> w/
{
  struct ListNode *p;
  p = 0;
  /*@ inv <heap> list(p)(7B), list(x)(7C) </heap> /\ A = rev(7B) @ 7C
  while(x) {
    struct ListNode *y;
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}

struct ListNode* append(struct ListNode *x, struct ListNode *y)
/*@ rule <<> $ => return x1; </>
  <heap> list(x)(A), list(y)(B) => list(x1)(A @ B) </heap> w/
{
  struct ListNode *p;
  if (x == 0)
    return y;
  p = x;
  /*@ inv <heap> lseg(x, p)(7A1), list(p)(7A2) </heap>
  /*@ inv /\ A = 7A1 @ 7A2 /\ ~(p = 0) /\ y = ly w/
  while (p->next)
    p = p->next;
  p->next = y;
  return x;
}

int length(struct ListNode *x)
/*@ rule <<> $ => return len(A); </> <heap> list(x)(A) </heap>
{
  int l;
  l = 0;
  /*@ inv <heap> lseg(old(x), x)(7A1), list(x)(7A2) </heap>
  /*@ inv /\ A = 7A1 @ 7A2 /\ l = len(7A1) w/
  while (x) {
    l += 1;
    x = x->next ;
  }
  return l;
}
```

```
Terminal — bash — 77x19
mobile-255-135:matchC andrei$ time (gcc demo/3-list/List3.c ; ./a.out ; rm a.
out )
x: 1 2 3 4 5
reverse(x): 5 4 3 2 1
x: 1 2 3
y: 1 2 3
append(x, y): 1 2 3 1 2 3

real    0m0.039s
user    0m0.022s
sys     0m0.014s
mobile-255-135:matchC andrei$ ./matchC demo/3-list/List3.c
Compiling program ... DONE! [0.476s]
Loading Maude ..... DONE! [0.963s]
Verifying program ... DONE! [0.129s]
Verification succeeded! [418155 rewrites, 12 feasible and 0 infeasible paths]
Output: skolem3, 7A
mobile-255-135:matchC andrei$
```

### Flattening a tree into a list

```
struct treeNode { int val; struct treeNode *left;
  struct treeNode *right; };
struct ListNode { int val; struct ListNode *next; };
struct stackNode { struct treeNode *val; struct stackNode *next; };

struct ListNode *toListIterative(struct treeNode *t)
/*@ rule <<> $ => return l1; </>
  <heap> tree(t)(T) => list(l1)(T) </heap> w/
{
  struct ListNode *l;
  struct stackNode *s;
  if (t == 0)
    return 0;

  l = 0;
  s = (struct stackNode *) malloc(sizeof(struct stackNode));
  s->val = t;
  s->next = 0;
  /*@ inv <heap> treeList(s)(7T5), list(l1)(7A) </heap>
  /*@ inv /\ tree2List(T) = treeList2List(rev(7T5)) @ 7A w/
  while (s != 0) {
    struct treeNode *tn;
    struct ListNode *ln;
    struct stackNode *sn;
    sn = s;
    s = s->next ;
    tn = sn->val;
    free(sn);
    if (tn->left != 0) {
      sn = (struct stackNode *) malloc(sizeof(struct stackNode));
      sn->val = tn->left;
      sn->next = s;
      s = sn;
    }
    if (tn->right != 0) {
      sn = (struct stackNode *) malloc(sizeof(struct stackNode));
      sn->val = tn;
      sn->next = s;
      s = sn;
    }
    ln = (struct ListNode *) malloc(sizeof(struct ListNode));
    ln->val = tn->val;
    ln->next = l;
    l = ln;
    tn->left = tn->right = 0;
  }
  else {
    ln = (struct ListNode *) malloc(sizeof(struct ListNode));
    ln->val = tn->val;
    ln->next = l;
    l = ln;
    free(tn);
  }
}
return l;
}
```

```
Terminal — bash — 77x19
mobile-255-135:matchC andrei$ time (gcc demo/4-binary-tree/binary_tree3.c ;
./a.out ; rm a.out )
l: 1 2 3 4 5 6 7
l: 1 2 3 4 5 6 7

real    0m0.041s
user    0m0.024s
sys     0m0.014s
mobile-255-135:matchC andrei$ ./matchC demo/4-binary-tree/binary_tree3.c
Compiling program ... DONE! [0.477s]
Loading Maude ..... DONE! [0.244s]
Verifying program ... DONE! [0.596s]
Verification succeeded! [382276 rewrites, 9 feasible and 31 infeasible paths]
Output: 1 2 3 4 5 6 7 1 2 3 4 5 6 7
mobile-255-135:matchC andrei$
```

### Stack inspection

```
void trusted(int n);
void untrusted(int n);
void any(int n);

void trusted(int n)
/*@ rule <<> $ => return; </> <stack> S </stack> <out>_epsilon => A </out>
  if n >= 10 /\ in(hd(ids(S)), {main, trusted}) w/
{
  printf("%d ", n);
  untrusted(n);
  if (n)
    trusted(n - 1);
}

void untrusted(int n)
/*@ rule <<> $ => return; </> <stack> S </stack> <out>_epsilon => A </out>
  if in(trusted, ids(S)) w/
{
  printf("%d ", -n);
  if (n)
    any(n - 1);
}

void any(int n)
{
  // untrusted(n);
  if (n > 10)
    // possible security violated if n < 10
    trusted(n - 1);
}

int main()
{
  trusted(5);
  any(5);
}
return 0;
}
```

```
Terminal — bash — 77x16
mobile-255-135:matchC andrei$ time (gcc demo/5-stack-inspection/stack_inspect
ion3.c ; ./a.out ; rm a.out )
5 4 3 2 1 0
real    0m0.032s
user    0m0.016s
sys     0m0.013s
mobile-255-135:matchC andrei$ ./matchC demo/5-stack-inspection/stack_inspecti
on3.c
Compiling program ... DONE! [0.324s]
Loading Maude ..... DONE! [0.175s]
Verifying program ... DONE! [0.129s]
Verification succeeded! [79182 rewrites, 7 feasible and 1 infeasible paths]
Output: skolem3, 7A
mobile-255-135:matchC andrei$
```

## Formal Semantics

### KERNELC—a C-like language

Train Florin Sirligă and Grigore Rosu  
University of Illinois at Urbana-Champaign

**Abstract**  
KERNELC is a non-trivial subset of the C language (including memory allocation and pointer arithmetic), which is used to exemplify several runtime analysis capabilities of K. definitions, as well as concurrency power and ease in defining and capturing related memory models.  
**Research based on KERNELC**  
KERNELC originated in the study of memory safety for C and first presented in the following paper:  
Grigore Rosu, Wolfram Schulte, and Train Florin Sirligă. *Runtime Verification of C Memory Safety*.  
Runtime Verification '07'N. Lecture Notes in Computer Science 5779, 132–151, 2009.  
Since then it has been expanded and used for expressing and verifying concurrency features and anomalies for both sequentially-consistent and relaxed memory models, as detailed in Chapter 3 of:

Train-Florin Sirligă. *A Rewriting Approach to Concurrent Programming Language Design and Semantics*.  
PhD Thesis, University of Illinois, December 2010

MODULE KERNELC-SYNTHAX  
IMPORTS K-LATEX-PL-4D-PL-INT

#### KERNELC Syntax

This module specifies the syntax of KERNELC. The syntax has been kept as close to the C syntax as possible to allow an exemplary class of C programs to be parsed and executed with the KERNELC definition. Nevertheless, the syntax is quite small, covering only 93 constructs of the C language.

#### Arithmetic expression

SYNTAX Exp ::= Exp \* Exp [int]  
| Id  
| & Id  
| Exp \*  
| Exp ++  
| Exp -  
| Exp /  
| Exp %  
| Exp < Exp [int]  
| Exp <= Exp [int]  
| Exp < Exp [int]  
| Exp <= Exp [int]

#### Logical operations

SYNTAX Exp ::= ! Exp  
| Exp && Exp  
| Exp || Exp  
| Exp ! Exp

#### Input/Output

For simplicity we syntactically restrict the printf and scanf to have only one, identifiable, argument. As the & operator is not part of the language, we opt for two versions of scanf, first for reading (local) variables and the other for reading into heap.

#### Memory allocation and addressing

Again, for simplicity we split out a fixed syntax for malloc, using the size of integers as a multiplication factor and the result to a integer pointer.

SYNTAX Exp ::= NULL  
| Pointer  
| (int \* malloc ( Exp \* sizeof (int) ) [int]  
| free ( Exp ) [int]  
| \* Exp [int]  
| Exp [ Exp ]

#### Assignment

We leave Exp in the left side to allow both assigning to variables and heap locations.

#### Function invocation

SYNTAX Exp ::= Id ( List(Exp) [ int\*2 ]  
| Id ()

#### Random

SYNTAX Exp ::= random()  
| \* random ( Exp ) [ int\*1 ]

#### Statements

SYNTAX Stmt ::= Exp ; [int\*1]  
| { List(Exp)  
| do { List(Exp) } while ( Exp ) Stmt  
| if ( Exp ) Stmt  
| while ( Exp ) Stmt  
| return Exp ; [int\*1]

#### Function declaration

SYNTAX Stmt ::= Decl\* List(Decl) ( List(Exp) )

#### Include pragmas

This is abusing the C syntax to allow splitting program into fragments (statement lists) which are then included one in another.

SYNTAX Stmt ::= #include Stmt\*  
SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt ::= Stmt\*  
SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*

SYNTAX Stmt\* ::= Stmt\* Stmt\*